Geometry3D

Release 0.1.0

Minghao Gou

Apr 29, 2020

CONTENTS

CHAPTER

ONE

ABOUT GEOMETRY3D

Geometry3D is a simple python computational geographics library written in python. This library focuses on the functions and lacks efficiency which might be improved in future version.

1.1 Core Features

- Basic 3D Geometries: Point, Line, Plane, Segment, Convex Polygen and Convex Polyhedron.
- Basic Attributes Of Geometries: length, area, volume.
- Basic Relationships And Operations Between Geometries: move, angle, parallel, orthogonal, intersection.
- Overload Build-In Functions Such As *__contains__*, *__hash__*, *__eq__*, *__neg__*.
- A Naive Renderer Using *matplotlib*.

1.2 Resources

- Documents: https://geometry3d.readthedocs.io/en/latest/
- Code: https://github.com/GouMinghao/Geometry3D

CHAPTER TWO

INSTALLATION

Note: Tested on Linux and Windows at the moment.

2.1 Prerequisites

It is assumed that you already have Python 3 installed. If you want graphic support, you need to manually install matplotlib.

2.2 System wide installation

You can install Geometry3D via pip:

```
$ pip install Geometry3D
```

Alternatively, you can install Geometry3D from source:

```
$ git clone http://github.com/GouMinghao/Geometry3D
$ cd Geometry3D/
$ sudo pip install .
# Alternative:
$ sudo python setup.py install
```

Note that the Python (or pip) version you use to install Geometry3D must match the version you want to use Geometry3D with.

2.3 Virtualenv installation

Geometry3D can be installed inside a virtualenv just like any other python package, though I suggest the use of virtualenvwrapper.

CHAPTER

THREE

TUTORIALS

3.1 Creating Geometries

3.1.1 Creating Point

Creating a Point using three cordinates:

```
>>> from Geometry3D import *
>>> pa = Point(1,2,3)
>>> pa
Point(1, 2, 3)
```

Creating a Point using a list of coordinates:

```
>>> pb = Point([2,4,3])
>>> pb
Point(2, 4, 3)
```

Specifically, special Point can be created using class function:

```
>>> o = origin()
>>> o
Point(0, 0, 0)
```

3.1.2 Creating Vector

Creating a Vector using three cordinates:

```
>>> from Geometry3D import *
>>> va = Vector(1,2,3)
>>> va
Vector(1, 2, 3)
```

Creating a Vector using two Points:

```
>>> pa = Point(1,2,3)
>>> pb = Point(2,3,1)
>>> vb = Vector(pa,pb)
>>> vb
Vector(1, 1, -2)
```

Creating a Vector using a list of coordinates:

```
>>> vc = Vector([1,2,4])
>>> vc
Vector(1, 2, 4)
```

Specifically, special Vectors can be created using class functions:

```
>>> x_unit_vector()
Vector(1, 0, 0)
>>> y_unit_vector()
Vector(0, 1, 0)
>>> z_unit_vector()
Vector(0, 0, 1)
```

3.1.3 Creating Line

Creating Line using two Points:

```
>>> from Geometry3D import *
>>> pa = Point(1,2,3)
>>> pb = Point(2,3,1)
>>> l = Line(pa,pb)
>>> l
Line(sv=Vector(1, 2, 3),dv=Vector(1, 1, -2))
```

Creating Line using two Vectors:

```
>>> va = Vector(1,2,3)
>>> vb = Vector(-1,-2,-1)
>>> l = Line(va,vb)
>>> l
Line(sv=Vector(1, 2, 3),dv=Vector(-1, -2, -1))
```

Creating Line using a Point and a Vector:

```
Line(sv=Vector(1, 2, 3),dv=Vector(-1, -2, -1))
>>> pa = Point(2,6,-2)
>>> v = Vector(2,0,4)
>>> l = Line(pa,v)
>>> l
Line(sv=Vector(2, 6, -2),dv=Vector(2, 0, 4))
```

Specifically, special Lines can be created using class functions:

```
>>> x_axis()
Line(sv=Vector(0, 0, 0),dv=Vector(1, 0, 0))
>>> y_axis()
Line(sv=Vector(0, 0, 0),dv=Vector(0, 1, 0))
>>> z_axis()
Line(sv=Vector(0, 0, 0),dv=Vector(0, 0, 1))
```

3.1.4 Creating Plane

Creating Plane using three Points:

```
>>> from Geometry3D import *
>>> p1 = origin()
>>> p2 = Point(1,0,0)
>>> p3 = Point(0,1,0)
>>> p = Plane(p1,p2,p3)
>>> p
Plane(Point(0, 0, 0), Vector(0, 0, 1))
```

Creating Plane using a Point and two Vectors:

```
>>> p1 = origin()
>>> v1 = x_unit_vector()
>>> v2 = z_unit_vector()
>>> p = Plane(p1,v1,v2)
>>> p
Plane(Point(0, 0, 0), Vector(0, -1, 0))
```

Creating Plane using a Point and a Vector:

```
>>> p1 = origin()
>>> p = Plane(p1,Vector(1,1,1))
>>> p
Plane(Point(0, 0, 0), Vector(1, 1, 1))
```

Creating Plane using four parameters:

```
# Plane(a, b, c, d):
# Initialise a plane given by the equation
# ax1 + bx2 + cx3 = d (general form).
>>> p = Plane(1,2,3,4)
>>> p
Plane(Point(-1.0, 1.0, 1.0), Vector(1, 2, 3))
```

Specifically, special Planes can be created using class functions:

```
>>> xy_plane()
Plane(Point(0, 0, 0), Vector(0, 0, 1))
>>> yz_plane()
Plane(Point(0, 0, 0), Vector(1, 0, 0))
>>> xz_plane()
Plane(Point(0, 0, 0), Vector(0, 1, 0))
```

3.1.5 Creating Segment

Creating Segment using two Points:

```
>>> from Geometry3D import *
>>> p1 = Point(0,0,2)
>>> p2 = Point(-1,2,0)
>>> s = Segment(p1,p2)
>>> s
Segment(Point(0, 0, 2), Point(-1, 2, 0))
```

Creating Segment using a Point and a Vector:

```
>>> s = Segment(origin(),x_unit_vector())
>>> s
Segment(Point(0, 0, 0), Point(1, 0, 0))
```

3.1.6 Creating ConvexPolygen

Creating ConvexPolygen using a tuple of points:

```
>>> from Geometry3D import *
>>> pa = origin()
>>> pb = Point(1,1,0)
>>> pc = Point(1,0,0)
>>> pd = Point(0,1,0)
>>> cpg = ConvexPolygen((pa,pb,pc,pd))
>>> cpg
ConvexPolygen((Point(0, 0, 0), Point(0, 1, 0), Point(1, 1, 0), Point(1, 0, 0)))
```

Specifically, Parallelogram can be created using one Point and two Vectors:

```
>>> pa = origin()
>>> cpg = Parallelogram(pa,x_unit_vector(),y_unit_vector())
>>> cpg
ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1, 0)))
```

3.1.7 Creating ConvexPolyhedron

Creating ConvexPolyhedron using a tuple of ConvexPolygens:

```
>>> from Geometry3D import *
>>> a = Point(1,1,1)
>>> b = Point(-1,1,1)
>>> c = Point(-1,-1,1)
>>> d = Point(1,-1,1)
>>> e = Point(1,1,-1)
>>> f = Point(-1,1,-1)
>>> g = Point(-1,-1,-1)
>>> h = Point(1,-1,-1)
>>> cpg0 = ConvexPolygen((a,d,h,e))
>>> cpg1 = ConvexPolygen((a,e,f,b))
>>> cpg2 = ConvexPolygen((c,b,f,g))
>>> cpg3 = ConvexPolygen((c,g,h,d))
>>> cpg4 = ConvexPolygen((a,b,c,d))
>>> cpg5 = ConvexPolygen((e,h,g,f))
>>> cph0 = ConvexPolyhedron((cpg0,cpg1,cpg2,cpg3,cpg4,cpg5))
>>> cph0
ConvexPolyhedron
pyramid set:{Pyramid(ConvexPolygen((Point(1, 1, -1), Point(1, -1, -1), Point(-1, -1, -
↪1), Point(-1, 1, -1))), Point(0.0, 0.0, 0.0)), Pyramid(ConvexPolygen((Point(1, 1,␣
↪1), Point(1, 1, -1), Point(-1, 1, -1), Point(-1, 1, 1))), Point(0.0, 0.0, 0.0)),␣
↪Pyramid(ConvexPolygen((Point(-1, -1, 1), Point(-1, 1, 1), Point(-1, 1, -1), Point(-
↪1, -1, -1))), Point(0.0, 0.0, 0.0)), Pyramid(ConvexPolygen((Point(-1, -1, 1),␣
↪Point(-1, -1, -1), Point(1, -1, -1), Point(1, -1, 1))), Point(0.0, 0.0, 0.0)),␣
↪Pyramid(ConvexPolygen((Point(1, 1, 1), Point(1, -1, 1), Point(1, -1, -1), Point(1,␣
↪1, -1))), Point(0.0, 0.0, 0.0)), Pyramid(ConvexPolygen((Point(1, 1, 1), Point(-1, 1,
↪ 1), Point(-1, -1, 1), Point(1, -1, 1))), Point(0.0, 0.0, 0.0))}
```

(continues on next page)

(continued from previous page)

```
point set:{Point(1, 1, -1), Point(-1, -1, -1), Point(1, -1, 1), Point(-1, 1, 1),␣
↪Point(1, 1, 1), Point(-1, -1, 1), Point(-1, 1, -1), Point(1, -1, -1)}
```

Specifically, Parallelepiped can be created using a Point and Three Vectors:

```
>>> cph = Parallelepiped(origin(),x_unit_vector(),y_unit_vector(),z_unit_vector())
>>> cph
ConvexPolyhedron
pyramid set:{Pyramid(ConvexPolygen((Point(1, 1, 1), Point(0, 1, 1), Point(0, 1, 0),␣
↪Point(1, 1, 0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(0, 0, 0),␣
↪Point(0, 1, 0), Point(0, 1, 1), Point(0, 0, 1))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 0, 1), Point(0, 0,␣
↪1))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1, 1, 1), Point(1, 0, 1),␣
↪Point(1, 0, 0), Point(1, 1, 0))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1,␣
↪0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1, 1, 1), Point(0, 1, 1),␣
↪Point(0, 0, 1), Point(1, 0, 1))), Point(0.5, 0.5, 0.5))}
point set:{Point(0, 0, 1), Point(1, 1, 1), Point(1, 1, 0), Point(0, 1, 1), Point(1, 0,
↪ 1), Point(0, 0, 0), Point(1, 0, 0), Point(0, 1, 0)}
```

3.2 Renderer Examples

3.2.1 Creating Geometries

```
>>> a = Point(1,2,1)
>>> c = Point(-1,-1,1)
>>> d = Point(1,-1,1)
>>> e = Point(1,1,-1)
>>> h = Point(1,-1,-1)
>>>
>>> s = Segment(a,c)
>>>
>>> cpg = ConvexPolygen((a,d,h,e))
>>>
>>> cph = Parallelepiped(Point(-1.5,-1.5,-1.5),Vector(2,0,0),Vector(0,2,0),Vector(0,0,
↪2))
```

3.2.2 Getting a Renderer

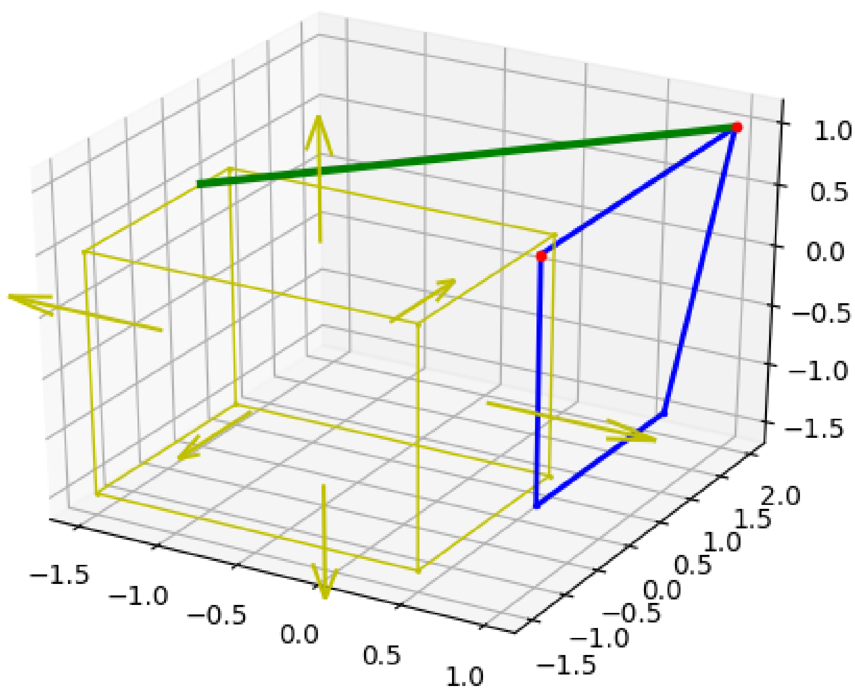
```
>>> r = Renderer(backend='matplotlib')
```

3.2.3 Adding Geometries

```
>>> r.add((a,'r',10),normal_length=0)
>>> r.add((d,'r',10),normal_length=0)
>>> r.add((s,'g',3),normal_length=0)
>>> r.add((cpg,'b',2),normal_length=0)
>>> r.add((cph,'y',1),normal_length=1)
```

3.2.4 Displaying Geometries

```
>>> r.show()
```



3.3 Getting Attributes

3.3.1 Creating Geometries

```
>>> a = Point(1,1,1)
>>> d = Point(1,-1,1)
>>> c = Point(-1,-1,1)
>>> e = Point(1,1,-1)
>>> h = Point(1,-1,-1)
>>>
>>> s = Segment(a,c)
>>>
>>> cpg = ConvexPolygen((a,d,h,e))
>>>
>>> cph = Parallelepiped(Point(-1,-1,-1),Vector(2,0,0),Vector(0,2,0),Vector(0,0,2))
```

3.3.2 Calculating the length

```
>>> s.length() # 2 * sqrt(2)
2.8284271247461903
>>> cpg.length() # 8
8.0
>>> cph.length() # 24
24.0
```

3.3.3 Calculating the area

```
>>> cph.area() # 24
23.999999999999993
>>> cpg.area() # 4
3.9999999999999982
>>> # Floating point calculation error
```

3.3.4 Calculating the volume

```
>>> cph.volume() # 8
7.999999999999995
>>> volume(cph0) # 8
7.999999999999995
```

3.4 Operations Examples

3.4.1 move

Move a Point:

```
>>> a = Point(1,2,1)
>>> print('a before move:{}'.format(a))
a before move:Point(1, 2, 1)
>>> a.move(x_unit_vector())
Point(2, 2, 1)
>>> print('a after move:{}'.format(a))
a after move:Point(2, 2, 1)
```

Move a Segment:

```
>>> b = origin()
>>> c = Point(1,2,3)
>>> s = Segment(b,c)
>>> s
Segment(Point(0, 0, 0), Point(1, 2, 3))
>>> s.move(Vector(-1,-2,-3))
Segment(Point(-1, -2, -3), Point(0, 0, 0))
>>> s
Segment(Point(-1, -2, -3), Point(0, 0, 0))
```

Move a ConvexPolygen **Without** Changing the Original Object:

```
>>> import copy
>>> cpg0 = Parallelogram(origin(),x_unit_vector(),y_unit_vector())
>>> cpg0
ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1, 0)))
>>> cpg1 = copy.deepcopy(cpg0).move(Vector(0,0,1))
>>> cpg0
ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1, 0)))
>>> cpg1
ConvexPolygen((Point(0, 0, 1), Point(1, 0, 1), Point(1, 1, 1), Point(0, 1, 1)))
```

3.4.2 Intersection

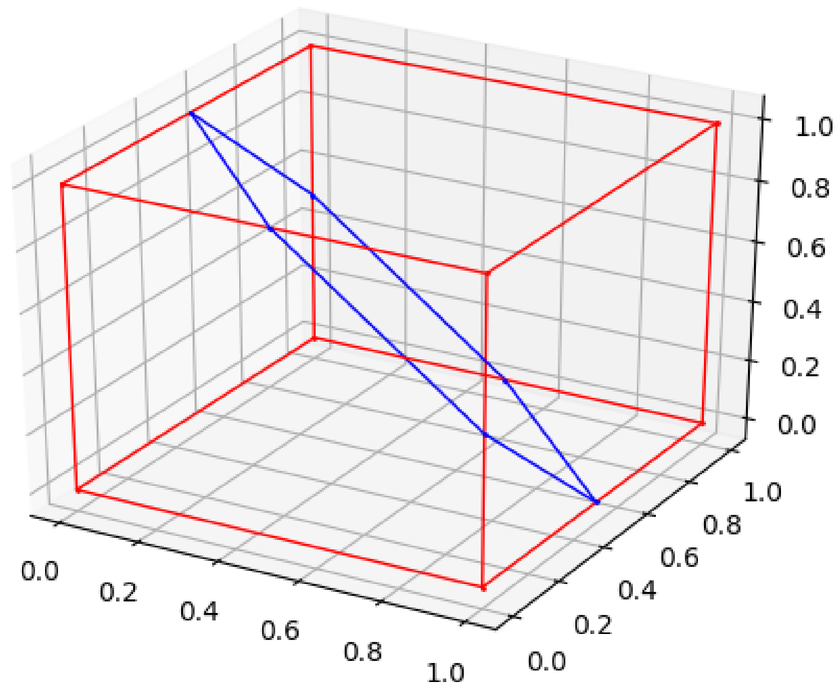
The operation of intersection is very complex. There are a total of 21 situations.

obj1	obj2	output obj
Point	Point	None, Point
Point	Line	None, Point
Point	Plane	None, Point
Point	Segment	None, Point
Point	ConvexPolygen	None, Point
Point	ConvexPolyhedron	None, Point
Line	Line	None, Point, Line
Line	Plane	None, Point, Line
Line	Segment	None, Point, Segment
Line	ConvexPolygen	None, Point, Segment
Line	ConvexPolyhedron	None, Point, Segment
Plane	Plane	None, Line, Plane
Plane	Segment	None, Point, Segment
Plane	ConvexPolygen	None, Point, Segment, ConvexPolygen
Plane	ConvexPolyhedron	None, Point, Segment, ConvexPolygen
Segment	Segment	None, Point, Segment
Segment	ConvexPolygen	None, Point, Segment
Segment	ConvexPolyhedron	None, Point, Segment
ConvexPolygen	ConvexPolygen	None, Point, Segment, ConvexPolygen
ConvexPolygen	ConvexPolyhedron	None, Point, Segment, ConvexPolygen
ConvexPolyhedron	ConvexPolyhedron	None, Point, Segment, ConvexPolygen, ConvexPolyhedron

All of the situations above are implemented. The documentation shows some examples.

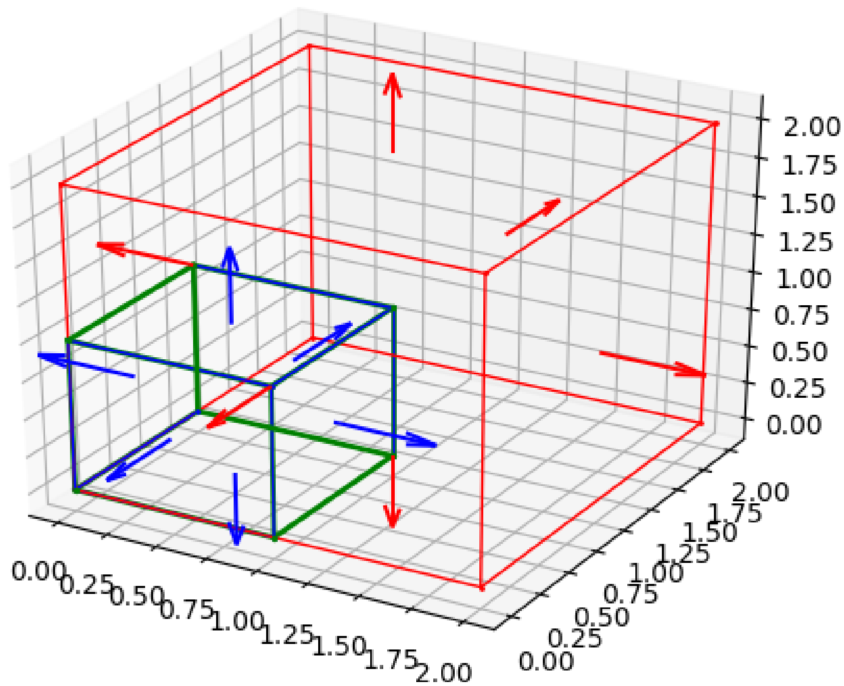
Example 1:

```
>>> po = origin()
>>> l1 = x_axis()
>>> l2 = y_axis()
>>> intersection(po,l1)
Point(0, 0, 0)
>>> intersection(l1,l2)
Point(0.0, 0.0, 0.0)
>>> s1 = Segment(Point(1,0,1),Point(0,1,1))
>>> s2 = Segment(Point(0,0,1),Point(1,1,1))
>>> s3 = Segment(Point(0.5,0.5,1),Point(-0.5,1.5,1))
>>> intersection(s1,s2)
Point(0.5, 0.5, 1.0)
>>> intersection(s1,s3)
Segment(Point(0.5, 0.5, 1.0), Point(0, 1, 1))
>>> intersection(l1,s1) is None
True
>>> cph0 = Parallelepiped(origin(),x_unit_vector(),y_unit_vector(),z_unit_vector())
>>> p = Plane(Point(0.5,0.5,0.5),Vector(1,1,1))
>>> cpg = intersection(cph0,p)
>>> r = Renderer()
>>> r.add((cph0,'r',1),normal_length = 0)
>>> r.add((cpg,'b',1),normal_length=0)
>>> r.show()
```



Example 2:

```
>>> from Geometry3D import *
>>> import copy
>>> r = Renderer()
>>> cph0 = Parallelepiped(origin(),x_unit_vector(),y_unit_vector(),z_unit_vector())
>>> cph6 = Parallelepiped(origin(),2 * x_unit_vector(),2 * y_unit_vector(),2 * z_unit_
↪vector())
>>> r.add((cph0,'b',1),normal_length = 0.5)
>>> r.add((cph6,'r',1),normal_length = 0.5)
>>> r.add((intersection(cph6,cph0),'g',2))
>>> print(intersection(cph0,cph6))
ConvexPolyhedron
pyramid set:{Pyramid(ConvexPolygen((Point(1, 1, 1), Point(0, 1, 1), Point(0.0, 0.0, 1.
↪0), Point(1, 0, 1))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1.0, 0.0,␣
↪0.0), Point(1, 0, 1), Point(1, 1, 1), Point(1, 1, 0))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(1, 1, 0), Point(1, 1, 1), Point(0, 1, 1), Point(0.0, 1.
↪0, 0.0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(0, 0, 1), Point(0, 0,
↪ 0), Point(0, 1, 0), Point(0, 1, 1))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(1, 0, 0), Point(1, 0, 1), Point(0, 0, 1), Point(0, 0,␣
↪0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1, 1, 0), Point(1, 0, 0),␣
↪Point(0, 0, 0), Point(0, 1, 0))), Point(0.5, 0.5, 0.5))}
point set:{Point(1, 1, 0), Point(1, 1, 1), Point(0, 0, 1), Point(0, 1, 0), Point(0, 1,
↪ 1), Point(1.0, 0.0, 0.0), Point(0, 0, 0), Point(1, 0, 1)}
>>> r.show()
```



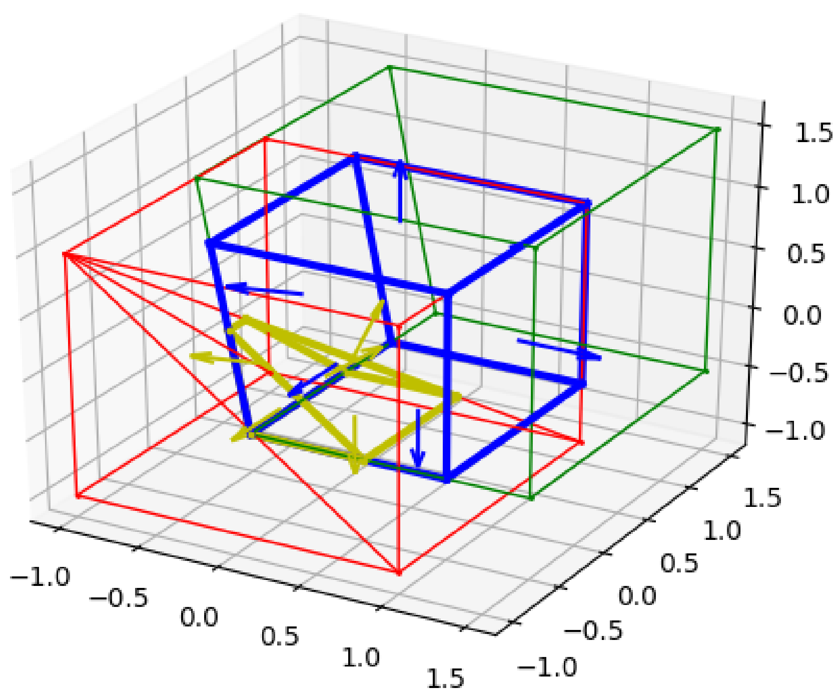
Example 3:

```
>>> from Geometry3D import *
>>>
>>> a = Point(1,1,1)
>>> b = Point(-1,1,1)
>>> c = Point(-1,-1,1)
>>> d = Point(1,-1,1)
>>> e = Point(1,1,-1)
>>> f = Point(-1,1,-1)
>>> g = Point(-1,-1,-1)
>>> h = Point(1,-1,-1)
>>> cph0 = Parallelepiped(Point(-1,-1,-1),Vector(2,0,0),Vector(0,2,0),Vector(0,0,2))
>>> cpg12 = ConvexPolygen((e,c,h))
>>> cpg13 = ConvexPolygen((e,f,c))
>>> cpg14 = ConvexPolygen((c,f,g))
>>> cpg15 = ConvexPolygen((h,c,g))
>>> cpg16 = ConvexPolygen((h,g,f,e))
>>> cph1 = ConvexPolyhedron((cpg12,cpg13,cpg14,cpg15,cpg16))
>>> a1 = Point(1.5,1.5,1.5)
>>> b1 = Point(-0.5,1.5,1.5)
>>> c1 = Point(-0.5,-0.5,1.5)
>>> d1 = Point(1.5,-0.5,1.5)
>>> e1 = Point(1.5,1.5,-0.5)
>>> f1 = Point(-0.2,1.5,-0.5)
>>> g1 = Point(-0.2,-0.5,-0.5)
```

(continues on next page)

(continued from previous page)

```
>>> h1 = Point(1.5,-0.5,-0.5)
>>>
>>> cpg6 = ConvexPolygen((a1,d1,h1,e1))
>>> cpg7 = ConvexPolygen((a1,e1,f1,b1))
>>> cpg8 = ConvexPolygen((c1,b1,f1,g1))
>>> cpg9 = ConvexPolygen((c1,g1,h1,d1))
>>> cpg10 = ConvexPolygen((a1,b1,c1,d1))
>>> cpg11 = ConvexPolygen((e1,h1,g1,f1))
>>> cph2 = ConvexPolyhedron((cpg6,cpg7,cpg8,cpg9,cpg10,cpg11))
>>> cph3 = intersection(cph0,cph2)
>>>
>>> cph4 = intersection(cph1,cph2)
>>> r = Renderer()
>>> r.add((cph0,'r',1),normal_length = 0)
>>> r.add((cph1,'r',1),normal_length = 0)
>>> r.add((cph2,'g',1),normal_length = 0)
>>> r.add((cph3,'b',3),normal_length = 0.5)
>>> r.add((cph4,'y',3),normal_length = 0.5)
>>> r.show()
```

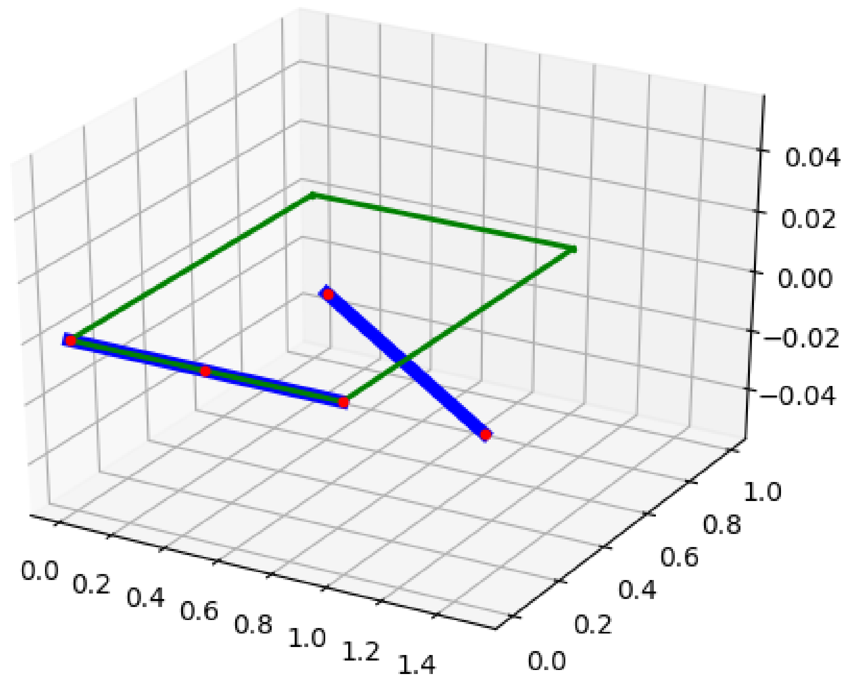


3.5 Build-In Functions

3.5.1 __contains__

__contains__ is used in build-in operator *in*, here are some examples:

```
>>> a = origin()
>>> b = Point(0.5,0,0)
>>> c = Point(1.5,0,0)
>>> d = Point(1,0,0)
>>> e = Point(0.5,0.5,0)
>>> s1 = Segment(origin(),d)
>>> s2 = Segment(e,c)
>>> a in s1
True
>>> b in s1
True
>>> c in s1
False
>>> a in s2
False
>>> b in s2
False
>>> c in s2
True
>>> cpg = Parallelogram(origin(),x_unit_vector(),y_unit_vector())
>>> a in cpg
True
>>> b in cpg
True
>>> c in cpg
False
>>> s1 in cpg
True
>>> s2 in cpg
False
>>>
>>> r=Renderer()
>>> r.add((a,'r',10))
>>> r.add((b,'r',10))
>>> r.add((c,'r',10))
>>> r.add((d,'r',10))
>>> r.add((e,'r',10))
>>> r.add((s1,'b',5))
>>> r.add((s2,'b',5))
>>> r.add((cpg,'g',2))
>>> r.show()
```



3.5.2 __*hash*___

__*hash*__ is used in set, here are some examples:

```
>>> a = set()
>>> a.add(origin())
>>> a
{Point(0, 0, 0)}
>>> a.add(Point(0,0,0))
>>> a
{Point(0, 0, 0)}
>>> a.add(Point(0,0,0.01))
>>> a
{Point(0, 0, 0), Point(0.0, 0.0, 0.01)}
>>>
>>> b = set()
>>> b.add(Segment(origin(),Point(1,0,0)))
>>> b
{Segment(Point(0, 0, 0), Point(1, 0, 0))}
>>> b.add(Segment(Point(1.0,0,0),Point(0,0,0)))
>>> b
{Segment(Point(0, 0, 0), Point(1, 0, 0))}
>>> b.add(Segment(Point(0,0,0),Point(0,1,1)))
>>> b
```

(continues on next page)

(continued from previous page)

```
{Segment(Point(0, 0, 0), Point(1, 0, 0)), Segment(Point(0, 0, 0), Point(0, 1, 1))}
```

3.5.3 *__eq__*

__eq__ is the build-in operator ==, here are some examples:

```
>>> a = origin()
>>> b = Point(1,0,0)
>>> c = Point(0,0,0)
>>> d = Point(2,0,0)
>>> a == b
False
>>> a == c
True
>>>
>>> s1 = Segment(a,b)
>>> s2 = Segment(a,b)
>>> s3 = Segment(b,a)
>>> s4 = Segment(a,d)
>>> s1 == s2
True
>>> s1 == s3
True
>>> s1 == s4
False
>>>
>>> cpg0 = ConvexPolygen((origin(),Point(1,0,0),Point(0,1,0),Point(1,1,0)))
>>> cpg1 = Parallelogram(origin(),x_unit_vector(),y_unit_vector())
>>> cpg0 == cpg1
True
```

3.5.4 *__neg__*

__neg__ is the build-in operator -, here are some examples:

```
>>> p = Plane(origin(),z_unit_vector())
>>> p
Plane(Point(0, 0, 0), Vector(0, 0, 1))
>>> -p
Plane(Point(0, 0, 0), Vector(0, 0, -1))
```

3.6 Dealing With Floating Numbers

There will be some errors in floating numbers computations. So identical objects may be deemed different. To tackle with this problem, this library believe two objects equal if their difference is smaller that a small number *eps*. Another value is named *significant number* has the relationship with eps:

```
significant number = -log(eps)
```

The default value of *eps* is 1e-10. You can access and change the value as follows:

```
>>> get_eps()
1e-10
>>> get_sig_figures()
10
>>> set_sig_figures(5)
>>> get_eps()
1e-05
>>> get_sig_figures()
5
>>> set_eps(1e-12)
>>> get_eps()
1e-12
>>> get_sig_figures()
12
```

CHAPTER

FOUR

PYTHON API

4.1 Geometry3D.calc package

4.1.1 Submodules

4.1.2 Geometry3D.calc.acute module

Acute Module

`Geometry3D.calc.acute.`**`acute`**`(`*rad*`)`

Input:

- rad: A angle in rad.

Output:

If the given angle is >90° (pi/2), return the opposite angle.

Return the angle else.

4.1.3 Geometry3D.calc.angle module

Angle Module

`Geometry3D.calc.angle.`**`angle`**`(`*a*, *b*`)`

Input:

- a: Line/Plane/Plane/Vector
- b: Line/Line/Plane/Vector

Output:

The angle (in radians) between

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

`Geometry3D.calc.angle.`**`parallel`**`(`*a*, *b*`)`

Input:

- a:Line/Plane/Plane/Vector

- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are parallel. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

`Geometry3D.calc.angle.`**`orthogonal`**`(`*a*, *b*`)`

Input:

- a:Line/Plane/Plane/Vector
- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are orthogonal. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

4.1.4 Geometry3D.calc.aux_calc module

Auxilary Calculation Module.

Auxilary calculation functions for calculating intersection

`Geometry3D.calc.aux_calc.`**`get_projection_length`**`(`*v1*, *v2*`)`

Input:

- v1: Vector
- v2: Vector

Output:

The length of vector that v1 projected on v2

`Geometry3D.calc.aux_calc.`**`get_relative_projection_length`**`(`*v1*, *v2*`)`

Input:

- v1: Vector
- v2: Vector

Output:

The ratio of length of vector that v1 projected on v2 and the length of v2

`Geometry3D.calc.aux_calc.`**`get_segment_from_point_list`**`(`*point_list*`)`

Input:

- point_list: a list of Points

Output:

The longest segment between the points

Geometry3D.calc.aux_calc.**get_segment_convexpolyhedron_intersection_point_set**(*s*, *cph*)

Input:

- s: Segment
- cph: ConvexPolyhedron

Output:

A set of intersection points

Geometry3D.calc.aux_calc.**get_segment_convexpolygen_intersection_point_set**(*s*, *cpg*)

Input:

- s: Segment
- cpg: ConvexPolygen

Output:

A set of intersection points

Geometry3D.calc.aux_calc.**points_in_a_line**(*points*)

Input:

- points: Tuple or list of Points

Output:

A set of intersection points

4.1.5 Geometry3D.calc.distance module

Distance Module

Geometry3D.calc.distance.**distance**(*a*, *b*)

Input:

- a: Point/Line/Line/Plane/Plane
- b: Point/Point/Line/Point/Line

Output:

Returns the distance between two objects. This includes

- Point/Point
- Line/Point
- Line/Line
- Plane/Point
- Plane/Line

4.1.6 Geometry3D.calc.intersection module

Intersection Module

`Geometry3D.calc.intersection.intersection(a, b)`

Input:

- a: GeoBody or None
- b: GeoBody or None

Output:

The Intersection.

Maybe None or GeoBody

4.1.7 Geometry3D.calc.volume module

Volume module

`Geometry3D.calc.volume.volume(arg)`

Input:

- arg: Pyramid or ConvexPolyhedron

Output:

Returns the object volume. This includes

- Pyramid
- ConvexPolyhedron

4.1.8 Module contents

`Geometry3D.calc.distance(a, b)`

Input:

- a: Point/Line/Line/Plane/Plane
- b: Point/Point/Line/Point/Line

Output:

Returns the distance between two objects. This includes

- Point/Point
- Line/Point
- Line/Line
- Plane/Point
- Plane/Line

`Geometry3D.calc.intersection(a, b)`

Input:

- a: GeoBody or None
- b: GeoBody or None

Output:

The Intersection.

Maybe None or GeoBody

`Geometry3D.calc.`**`parallel`**`(`*a*, *b*`)`

Input:

- a:Line/Plane/Plane/Vector
- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are parallel. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

`Geometry3D.calc.`**`angle`**`(`*a*, *b*`)`

Input:

- a: Line/Plane/Plane/Vector
- b: Line/Line/Plane/Vector

Output:

The angle (in radians) between

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

`Geometry3D.calc.`**`orthogonal`**`(`*a*, *b*`)`

Input:

- a:Line/Plane/Plane/Vector
- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are orthogonal. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

`Geometry3D.calc.`**`volume`**`(`*arg*`)`

Input:

- arg: Pyramid or ConvexPolyhedron

Output:

Returns the object volume. This includes

- Pyramid
- ConvexPolyhedron

`Geometry3D.calc.`**`get_projection_length`**(*v1*, *v2*)

Input:

- v1: Vector
- v2: Vector

Output:

The length of vector that v1 projected on v2

`Geometry3D.calc.`**`get_relative_projection_length`**(*v1*, *v2*)

Input:

- v1: Vector
- v2: Vector

Output:

The ratio of length of vector that v1 projected on v2 and the length of v2

`Geometry3D.calc.`**`get_segment_from_point_list`**(*point_list*)

Input:

- point_list: a list of Points

Output:

The longest segment between the points

`Geometry3D.calc.`**`get_segment_convexpolyhedron_intersection_point_set`**(*s*, *cph*)

Input:

- s: Segment
- cph: ConvexPolyhedron

Output:

A set of intersection points

`Geometry3D.calc.`**`get_segment_convexpolygen_intersection_point_set`**(*s*, *cpg*)

Input:

- s: Segment
- cpg: ConvexPolygen

Output:

A set of intersection points

`Geometry3D.calc.`**`points_in_a_line`**(*points*)

Input:

- points: Tuple or list of Points

Output:

A set of intersection points

4.2 Geometry3D.geometry package

4.2.1 Submodules

4.2.2 Geometry3D.geometry.body module

Geobody module

class Geometry3D.geometry.body.**GeoBody**

Bases: `object`

A base class for geometric objects that provides some common methods to work with. In the end, everything is dispatched to Geometry3D.calc.calc.* anyway, but it sometimes feels nicer to write it like L1.intersection(L2) instead of intersection(L1, L2)

angle(*other*)
: return the angle between self and other

distance(*other*)
: return the distance between self and other

intersection(*other*)
: return the intersection between self and other

orthogonal(*other*)
: return if self and other are orthogonal to each other

parallel(*other*)
: return if self and other are parallel to each other

4.2.3 Geometry3D.geometry.line module

Line Module

class Geometry3D.geometry.line.**Line**(*a*, *b*)

Bases: *Geometry3D.geometry.body.GeoBody*

- Line(Point, Point):

A Line going through both given points.

- Line(Point, Vector):

A Line going through the given point, in the direction pointed by the given Vector.

- Line(Vector, Vector):

The same as Line(Point, Vector), but with instead of the point only the position vector of the point is given.

class_level = 1

move(*v*)
: Return the line that you get when you move self by vector v, self is also moved

parametric()

Returns (s, u) so that you can build the equation for the line _ _ _

g: x = s + ru ; r e R

classmethod x_axis()
: return x axis which is a Line

classmethod y_axis()
return y axis which is a Line

classmethod z_axis()
return z axis which is a Line

Geometry3D.geometry.line.**x_axis**()
return x axis which is a Line

Geometry3D.geometry.line.**y_axis**()
return y axis which is a Line

Geometry3D.geometry.line.**z_axis**()
return z axis which is a Line

4.2.4 Geometry3D.geometry.plane module

Plane module

class Geometry3D.geometry.plane.**Plane**(**args*)
Bases: *Geometry3D.geometry.body.GeoBody*

- Plane(Point, Point, Point):

Initialise a plane going through the three given points.

- Plane(Point, Vector, Vector):

Initialise a plane given by a point and two vectors lying on the plane.

- Plane(Point, Vector):

Initialise a plane given by a point and a normal vector (point normal form)

- Plane(a, b, c, d):

Initialise a plane given by the equation ax1 + bx2 + cx3 = d (general form).

class_level = 2

general_form()
Returns (a, b, c, d) so that you can build the equation

E: ax1 + bx2 + cx3 = d

to describe the plane.

move(*v*)
Return the plane that you get when you move self by vector v, self is also moved

parametric()

Returns (u, v, w) so that you can build the equation _ _ _ _

E: x = u + rv + sw ; (r, s) e R

to describe the plane (a point and two vectors).

point_normal()

Returns (p, n) so that you can build the equation _ _

E: (x - p) n = 0

to describe the plane.

classmethod xy_plane()
: return xy plane which is a Plane

classmethod xz_plane()
: return xz plane which is a Plane

classmethod yz_plane()
: return yz plane which is a Plane

Geometry3D.geometry.plane.**xy_plane**()
: return xy plane which is a Plane

Geometry3D.geometry.plane.**yz_plane**()
: return yz plane which is a Plane

Geometry3D.geometry.plane.**xz_plane**()
: return xz plane which is a Plane

4.2.5 Geometry3D.geometry.point module

Point Module

class Geometry3D.geometry.point.**Point**(**args*)

Bases: object

- Point(a, b, c)
- Point([a, b, c]):

The point with coordinates (a | b | c)

- Point(Vector):

The point that you get when you move the origin by the given vector. If the vector has coordinates (a | b | c), the point will have the coordinates (a | b | c) (as easy as pi).

class_level = 0

distance(*other*)
: Return the distance between self and other

move(*v*)
: Return the point that you get when you move self by vector v, self is also moved

classmethod origin()
: Returns the Point (0 | 0 | 0)

pv()
: Return the position vector of the point.

Geometry3D.geometry.point.**origin**()
: Returns the Point (0 | 0 | 0)

4.2.6 Geometry3D.geometry.polygen module

Polygen Module

class Geometry3D.geometry.polygen.**ConvexPolygen**(*pts*, *reverse=False*, *check_convex=False*)

Bases: *Geometry3D.geometry.body.GeoBody*

- ConvexPolygens(points)

points: a tuple of points.

The points needn't to be in order.

The convexity should be guaranteed. This function **will not** check the convexity. If the Polygen is not convex, there might be errors.

classmethod Parallelogram(*base_point*, *v1*, *v2*)

A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

area()

Input:

- self

Output:

- The area of the convex polygen

class_level = 4

eq_with_normal(*other*)

return whether self equals with other considering the normal

hash_with_normal()

return the hash value considering the normal

in_(*other*)

Input:

- self: ConvexPolygen
- other: Plane

Output:

- whether self in other

length()

return the total length of ConvexPolygen

move(*v*)

Return the ConvexPolygen that you get when you move self by vector v, self is also moved

segments()

Input:

- self

Output:

- iterator of segments

`Geometry3D.geometry.polygen.`**`Parallelogram`**(*base_point*, *v1*, *v2*)

A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

4.2.7 Geometry3D.geometry.polyhedron module

Polyhedron Module

`class` `Geometry3D.geometry.polyhedron.`**`ConvexPolyhedron`**(*convex_polygens*)

Bases: *`Geometry3D.geometry.body.GeoBody`*

`classmethod Parallelepiped`(*base_point*, *v1*, *v2*, *v3*)

A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

`area`()

return the total area of the polyhedron

`class_level = 5`

Input:

- convex_polygens: tuple of ConvexPolygens

Output:

- ConvexPolyhedron
- The correctness of convex_polygens are checked According to Euler's formula.
- The normal of the convex polygens are checked and corrected which should be toward the outer direction

`length`()

return the total length of the polyhedron

`move`(*v*)

Return the ConvexPolyhedron that you get when you move self by vector v, self is also moved

`volume`()

return the total volume of the polyhedron

`Geometry3D.geometry.polyhedron.`**`Parallelepiped`**(*base_point*, *v1*, *v2*, *v3*)

A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

4.2.8 Geometry3D.geometry.pyramid module

Pyramid Module

class `Geometry3D.geometry.pyramid.`**`Pyramid`**(*cp*, *p*, *direct_call=True*)

Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- cp: a ConvexPolygen
- p: a Point

`height`()

return the height of the pyramid

`volume`()

return the volume of the pryamid

4.2.9 Geometry3D.geometry.segment module

Segment Module

class `Geometry3D.geometry.segment.`**`Segment`**(*a*, *b*)

Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- Segment(Point,Point)
- Segment(Point,Vector)

`class_level = 3`

`in_`(*other*)

other can be plane or line

`length`()

retutn the length of the segment

`move`(*v*)

Return the Segment that you get when you move self by vector v, self is also moved

`parametric`()

Returns (start_point, end_point) so that you can build the information for the segment

4.2.10 Module contents

class Geometry3D.geometry.**ConvexPolyhedron**(*convex_polygens*)

Bases: *Geometry3D.geometry.body.GeoBody*

classmethod Parallelepiped(*base_point*, *v1*, *v2*, *v3*)

A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

area()

return the total area of the polyhedron

class_level = 5

Input:

- convex_polygens: tuple of ConvexPolygens

Output:

- ConvexPolyhedron
- The correctness of convex_polygens are checked According to Euler's formula.
- The normal of the convex polygens are checked and corrected which should be toward the outer direction

length()

return the total length of the polyhedron

move(*v*)

Return the ConvexPolyhedron that you get when you move self by vector v, self is also moved

volume()

return the total volume of the polyhedron

Geometry3D.geometry.**Parallelepiped**(*base_point*, *v1*, *v2*, *v3*)

A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

class Geometry3D.geometry.**ConvexPolygen**(*pts*, *reverse=False*, *check_convex=False*)

Bases: *Geometry3D.geometry.body.GeoBody*

- ConvexPolygens(points)

points: a tuple of points.

The points needn't to be in order.

The convexity should be guaranteed. This function **will not** check the convexity. If the Polygen is not convex, there might be errors.

classmethod Parallelogram(*base_point*, *v1*, *v2*)
A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

area()

Input:

- self

Output:

- The area of the convex polygen

class_level = 4

eq_with_normal(*other*)
return whether self equals with other considering the normal

hash_with_normal()
return the hash value considering the normal

in_(*other*)

Input:

- self: ConvexPolygen
- other: Plane

Output:

- whether self in other

length()
return the total length of ConvexPolygen

move(*v*)
Return the ConvexPolygen that you get when you move self by vector v, self is also moved

segments()

Input:

- self

Output:

- iterator of segments

Geometry3D.geometry.**Parallelogram**(*base_point*, *v1*, *v2*)
A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

class Geometry3D.geometry.**Pyramid**(*cp*, *p*, *direct_call=True*)

Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- cp: a ConvexPolygen
- p: a Point

height()

return the height of the pyramid

volume()

return the volume of the pryamid

class Geometry3D.geometry.**Segment**(*a*, *b*)

Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- Segment(Point,Point)
- Segment(Point,Vector)

class_level = 3

in_(*other*)

other can be plane or line

length()

retutn the length of the segment

move(*v*)

Return the Segment that you get when you move self by vector v, self is also moved

parametric()

Returns (start_point, end_point) so that you can build the information for the segment

class Geometry3D.geometry.**Line**(*a*, *b*)

Bases: *Geometry3D.geometry.body.GeoBody*

- Line(Point, Point):

A Line going through both given points.

- Line(Point, Vector):

A Line going through the given point, in the direction pointed by the given Vector.

- Line(Vector, Vector):

The same as Line(Point, Vector), but with instead of the point only the position vector of the point is given.

class_level = 1

move(*v*)

Return the line that you get when you move self by vector v, self is also moved

parametric()

Returns (s, u) so that you can build the equation for the line _ _ _

g: x = s + ru ; r e R

classmethod x_axis()
return x axis which is a Line

classmethod y_axis()
return y axis which is a Line

classmethod z_axis()
return z axis which is a Line

class Geometry3D.geometry.**Plane**(**args*)
Bases: `Geometry3D.geometry.body.GeoBody`

- Plane(Point, Point, Point):

Initialise a plane going through the three given points.

- Plane(Point, Vector, Vector):

Initialise a plane given by a point and two vectors lying on the plane.

- Plane(Point, Vector):

Initialise a plane given by a point and a normal vector (point normal form)

- Plane(a, b, c, d):

Initialise a plane given by the equation ax1 + bx2 + cx3 = d (general form).

class_level = 2

general_form()
Returns (a, b, c, d) so that you can build the equation

E: ax1 + bx2 + cx3 = d

to describe the plane.

move(*v*)
Return the plane that you get when you move self by vector v, self is also moved

parametric()

Returns (u, v, w) so that you can build the equation _ _ _ _

E: x = u + rv + sw ; (r, s) e R

to describe the plane (a point and two vectors).

point_normal()

Returns (p, n) so that you can build the equation _ _

E: (x - p) n = 0

to describe the plane.

classmethod xy_plane()
return xy plane which is a Plane

classmethod xz_plane()
return xz plane which is a Plane

classmethod yz_plane()
return yz plane which is a Plane

class Geometry3D.geometry.**Point**(**args*)
Bases: `object`

- Point(a, b, c)
- Point([a, b, c]):

The point with coordinates (a | b | c)

- Point(Vector):

The point that you get when you move the origin by the given vector. If the vector has coordinates (a | b | c), the point will have the coordinates (a | b | c) (as easy as pi).

class_level = 0

distance(*other*)
Return the distance between self and other

move(*v*)
Return the point that you get when you move self by vector v, self is also moved

classmethod origin()
Returns the Point (0 | 0 | 0)

pv()
Return the position vector of the point.

`Geometry3D.geometry.`**origin**()
Returns the Point (0 | 0 | 0)

`Geometry3D.geometry.`**x_axis**()
return x axis which is a Line

`Geometry3D.geometry.`**y_axis**()
return y axis which is a Line

`Geometry3D.geometry.`**z_axis**()
return z axis which is a Line

`Geometry3D.geometry.`**xy_plane**()
return xy plane which is a Plane

`Geometry3D.geometry.`**yz_plane**()
return yz plane which is a Plane

`Geometry3D.geometry.`**xz_plane**()
return xz plane which is a Plane

4.3 Geometry3D.render package

4.3.1 Submodules

4.3.2 Geometry3D.render.arrow module

Arrow Module for Renderer

class `Geometry3D.render.arrow.`**Arrow**(*x, y, z, u, v, w, length*)
Bases: `object`

Arrow Class

get_tuple()
return the tuple expression of the arrow

4.3.3 Geometry3D.render.renderer module

Abstract Renderer Module

`Geometry3D.render.renderer.`**`Renderer`**(*backend='matplotlib'*)

Input:

- backend: the backend of the renderer

Only matplotlib is supported till now

4.3.4 Geometry3D.render.renderer_matplotlib module

Matplotlib Renderer Module

`class` `Geometry3D.render.renderer_matplotlib.`**`MatplotlibRenderer`**

Bases: `object`

Renderer module to visualize geometries

`add`(*obj*, *normal_length=0*)

Input:

- obj: a tuple (object,color,size)
- normal_length: the length of normal arrows for ConvexPolyhedron.

For other objects, normal_length should be zero. If you don't want to show the normal arrows for a ConvexPolyhedron, you can set normal_length to 0.

object can be Point, Segment, ConvexPolygen or ConvexPolyhedron

`show`()

Draw the image

4.3.5 Module contents

`Geometry3D.render.`**`Renderer`**(*backend='matplotlib'*)

Input:

- backend: the backend of the renderer

Only matplotlib is supported till now

4.4 Geometry3D.utils package

4.4.1 Submodules

4.4.2 Geometry3D.utils.constant module

Constant module

EPS and significant numbers for comparing float point numbers.

Two float numbers are deemed equal if they equal with each other within significant numbers.

Significant numbers = log(1 / eps) all the time

`Geometry3D.utils.constant.`**`set_eps`**`(`*eps=1e-10*`)`

Input:

- eps: floating number with 1e-10 the default

Output:

No output but set EPS to eps

Signigicant numbers is also changed.

`Geometry3D.utils.constant.`**`get_eps`**`()`

Input:

no input

Output:

- current eps: float

`Geometry3D.utils.constant.`**`get_sig_figures`**`()`

Input:

no input

Output:

- current significant numbers: int

`Geometry3D.utils.constant.`**`set_sig_figures`**`(`*sig_figures=10*`)`

Input:

- sig_figures: int with 10 the default

Output:

No output but set signigicant numbers to sig_figures

EPS is also changed.

4.4.3 Geometry3D.utils.logger module

Logger Module

`Geometry3D.utils.logger.`**`change_main_logger`**`()`

`Geometry3D.utils.logger.`**`get_main_logger`**`()`

Input:

No Input

Output:

main_logger: The logger instance

`Geometry3D.utils.logger.`**`set_log_level`**`(`*level='WARNING'*`)`

Input:

- level: a string of log level among ‘DEBUG’, ‘INFO’, ‘WARNING’, ‘ERROR’, ‘CRITICAL’.

    ‘WARNING’ is the default.

Output:

No output but setup the log level for the logger

4.4.4 Geometry3D.utils.solver module

Solver Module, An Auxilary Module

class `Geometry3D.utils.solver.`**`Solution`**(*s*)

Bases: `object`

Holds a solution to a system of equations.

`Geometry3D.utils.solver.`**`count`**(*f*, *l*)

`Geometry3D.utils.solver.`**`find_pivot_row`**(*m*)

`Geometry3D.utils.solver.`**`first_nonzero`**(*r*)

`Geometry3D.utils.solver.`**`gaussian_elimination`**(*m*)

Return the row echelon form of m by applying the gaussian elimination

`Geometry3D.utils.solver.`**`index`**(*f*, *l*)

`Geometry3D.utils.solver.`**`null`**(*f*)

`Geometry3D.utils.solver.`**`nullrow`**(*r*)

`Geometry3D.utils.solver.`**`shape`**(*m*)

`Geometry3D.utils.solver.`**`solve`**(*matrix*)

4.4.5 Geometry3D.utils.util module

Util Module

`Geometry3D.utils.util.`**`unify_types`**(*items*)

Promote all items to the same type. The resulting type is the "most valueable" that an item already has as defined by the list (top = least valueable):

- int
- float
- decimal.Decimal
- fractions.Fraction
- user defined

4.4.6 Geometry3D.utils.vector module

Vector Module

class `Geometry3D.utils.vector.`**`Vector`**(**args*)

Bases: `object`

Vector Class

`angle`(*other*)

Returns the angle (in radians) enclosed by both vectors.

`cross`(*other*)

Calculates the cross product of two vectors, defined as _ _ / x2y3 - x3y2 x × y = | x3y1 - x1y3 |

x1y2 - x2y1 /

The cross product is orthogonal to both vectors and its length is the area of the parallelogram given by x and y.

length()
Returns |v|, the length of the vector.

normalized()
Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

orthogonal(*other*)
Returns true if the two vectors are orthogonal

parallel(*other*)
Returns true if both vectors are parallel.

unit()
Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

classmethod x_unit_vector()
Returns the unit vector (1 | 0 | 0)

classmethod y_unit_vector()
Returns the unit vector (0 | 1 | 0)

classmethod z_unit_vector()
Returns the unit vector (0 | 0 | 1)

classmethod zero()
Returns the zero vector (0 | 0 | 0)

Geometry3D.utils.vector.**x_unit_vector**()
Returns the unit vector (1 | 0 | 0)

Geometry3D.utils.vector.**y_unit_vector**()
Returns the unit vector (0 | 1 | 0)

Geometry3D.utils.vector.**z_unit_vector**()
Returns the unit vector (0 | 0 | 1)

4.4.7 Module contents

Geometry3D.utils.**solve**(*matrix*)

class Geometry3D.utils.**Vector**(**args*)
Bases: object

Vector Class

angle(*other*)
Returns the angle (in radians) enclosed by both vectors.

cross(*other*)
Calculates the cross product of two vectors, defined as _ _ / x2y3 - x3y2 x × y = | x3y1 - x1y3 |

x1y2 - x2y1 /

The cross product is orthogonal to both vectors and its length is the area of the parallelogram given by x and y.

length()
Returns |v|, the length of the vector.

normalized()
Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

orthogonal(*other*)
Returns true if the two vectors are orthogonal

parallel(*other*)
Returns true if both vectors are parallel.

unit()
Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

classmethod x_unit_vector()
Returns the unit vector (1 | 0 | 0)

classmethod y_unit_vector()
Returns the unit vector (0 | 1 | 0)

classmethod z_unit_vector()
Returns the unit vector (0 | 0 | 1)

classmethod zero()
Returns the zero vector (0 | 0 | 0)

Geometry3D.utils.**x_unit_vector**()
Returns the unit vector (1 | 0 | 0)

Geometry3D.utils.**y_unit_vector**()
Returns the unit vector (0 | 1 | 0)

Geometry3D.utils.**z_unit_vector**()
Returns the unit vector (0 | 0 | 1)

Geometry3D.utils.**set_eps**(*eps=1e-10*)
Input:

- eps: floating number with 1e-10 the default

Output:

No output but set EPS to eps

Signigicant numbers is also changed.

Geometry3D.utils.**get_eps**()
Input:

no input

Output:

- current eps: float

Geometry3D.utils.**get_sig_figures**()
Input:

no input

Output:

- current significant numbers: int

Geometry3D.utils.**set_sig_figures**(*sig_figures=10*)
Input:

- sig_figures: int with 10 the default

Output:

No output but set signigicant numbers to sig_figures

EPS is also changed.

`Geometry3D.utils.set_log_level(level='WARNING')`

Input:

- level: a string of log level among 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'.

  'WARNING' is the default.

Output:

No output but setup the log level for the logger

`Geometry3D.utils.get_main_logger()`

Input:

No Input

Output:

main_logger: The logger instance

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

G

H

I

L

M

N

O

P

R

S

U

V

X

Y

Z